# Software Tech News

*The DoD Source for Software Technology Information.*

*Vol. 3- No. 2*
**Software Testing Part 1**

**In This Issue:**

Read additional Software Testing materials at:

www.dacs.dtic.mil/awareness/newsletters/listing.shtml

## Newsletter Series: Software Testing

**This is the first in a series of newsletters devoted to Software Testing.**

The Software Tech News will periodically devote an issue to this important topic.

### Call for Articles:

Potential authors may submit an article on Software Testing for publication in a future issue of the Software Tech News through one of the following methods:

**E-mail:** ldean@dacs.dtic.mil
**Fax** (315) 334-4964
**Mail:** DoD Data & Analysis Center for Software
Attn: Software Tech News Editor
775 Daedalian Drive
Rome, NY 13441-4909

**Any questions contact:**
Lon R. Dean - Editor
ldean@dacs.dtic.mil
(800) 214-7921

### A sampling of the articles scheduled to appear in Software Testing Part 2 include:

Testing Software Based Systems: The Final Frontier, Thomas Drake, Coastal Research and Technology

Using Models for Test Generation, Mark Blackburn, Software Productivity Consortium

Task-Based Software Testing, Daniel Telford, MacAulay Brown

Thread Based Integration Testing: Lessons Learned from an Iterative Approach, William Borgia and Neil Hrdlick, Northrup Grumman

DACS
DoD Data & Analysis Center for Software
http://www.dacs.dtic.mil

**Software Testing Series: Part 1**

# Testing Software: Challenges for the Future

*by Don Reifer, Reifer Consultants, Inc.*

## Introduction

When I hear the word testing, I become stressed. What a headache, I think. Pictures flash through my head of teams burning the midnight oil trying to get a product out. Ed Yourdon's book *Death March*[1] comes to mind along with visions of sleepless nights and troublesome days. Thoughts about budget and schedule problems clutter my head along with the seemingly ever-present performance problems that always seem to arise near the end of the project. I ponder, "how did we get in so much trouble?" and contemplate "have things gotten any better?"

When you give these questions some thought, you realize that we have made great strides in the realm of testing over the past decade, including shifts to incremental development and delivery that address persistent integration problems. Test management processes have matured and people seem to be paying attention to test issues earlier in their programs. Test methods and tools that we talked about just a decade ago are currently being used and working. Most important, we no longer seem to be beating our heads against the wall as we try to cope with the issues and challenges that seem to pop up the moment we start integrating and testing our products.

## Test Technology Has Moved Ahead

Let's look at the improvements by summarizing how we've dealt with the problems that existed just a decade ago. Then, let's look to the future to identify the challenges that testing must face in the near-term.

To check out where we've made progress, I've consulted some friends, old and new, in my bookcase and periodicals rack[2]. For example, I reviewed Fred Brooks's ***Mythical Man-Month***[3], Tom DeMarco's ***Controlling Software Projects***[4], Bob Glass' ***Recollections of Software Pioneers***[5], Walker Royce's excellent new book ***Software Project Management***[6] and others to see what they said on the subject of testing. They, and the test books that I have reviewed, seem to agree with the following observations relative to the test wisdom that is summarized in Table 1.

## New Challenges, Old Problems

Not only have we made progress in the world of testing, we've started tackling a host of new challenges. The world of software development is undergoing a lot of change these days with the influx of new paradigms (spiral, incremental development, drop and ship, etc.), technology (Java, active agents, etc.) and component-based software development. This change has fostered new approaches to evaluating and qualifying software components. For example, agents are now being used as part of several modern development environments to capture metrics data automatically. These metrics

**Table 1 Coping with Test Problems**

| No. | Problem | Current Wisdom |
|---|---|---|
| 1 | Testing considered late in the project | Start test planning and preparation the day that you start the project |
| 2 | Requirements not testable | Validate testability of requirements as you write the specification[7] |
| 3 | Integrate after all components have been thoroughly tested | Build a little, then test a little. Don't wait until the last moment to test. Try before you buy. |
| 4 | One step forward, two steps backward | Use repeatable processes to order the manner in which you integrate and test the system |
| 5 | Regression testing done ad hoc | Automate the test process and use tools to specify, perform and administer the conduct[8] |
| 6 | Test progress hard to measure (Test until you and/or your budget is exhausted) | Use a variety of standard software metrics to determine whether you have tested enough[9] |

may trigger actions when error rates and other indicators show quality goals are not being realized. Table 2 identifies some of the new challenges the test community faces and summarizes how they are trying to address them.

That's great news, you're probably thinking. We've got the test demon under control. Well, that's not exactly the case. Only leading firms within the industry has put these concepts to work systematically, repeatedly and consistently. The major reason behind this gap between theory and practice is simple, people buckle under deadline pressures. Anyone who's been there understands the problem. Software projects tend to get into trouble a little at a time, not all at once. As things go awry, process improvements, disciplined methods and other good ideas are discarded as efforts are made to stay on schedule. So, it seems we still face the same issues we did a decade ago even in light of the progress we have made. Simply stated, when we become entangled in the "crunch mode," testing discipline seems to go out the door.

Test research suffers similar maladies. There still is a push to improve specification technology. The motivation is to get rid of errors early and eliminate the need to test. While philosophically appealing, we still haven't figured out how to tame the specification beast. This emphasizes the need for research to address the test issues identified in Table 2. Unfortunately, the university and research community is not addressing this need. When you review the premier research programs in the United State and abroad, you see that most of their money is being spent on development rather than test topics.

Are my conclusions relative to progress within the testing field still valid? Let me answer this question by posing some questions. What would your management do when faced with a potential schedule slip? Would they have the guts to delay shipment because they are worried about poor product quality? Would they accept the risks inherent in deferring documenting the test results and getting their regression tests in order until after the delivery were made? How would they handle the situation when the user is screaming for results, members of the team are transitioning to new projects, and everyone involved seems overstressed, overworked and tired? What would you do if you were placed in their shoes?

## Opportunities and Dilemmas

In spite of the advances we seem to have made in the technology, we can conclude that the same pressures to release prematurely persist when it comes to testing. Perhaps, this is an important message. It says to me that we may need to alter the path we take as we embark on our quest for new and better ways to handle the test challenges I've outlined.

*Continued on page 4*

**Table 2. Addressing New Test Challenges**

| No. | Challenge | Current Solution Approach |
|---|---|---|
| 1 | Incremental and spiral paradigms[10] | Incremental testing; regression test baseline; early user testing (hopefully with prototype); use cases to define threads through software per usage views |
| 2 | COTS-based development paradigm[11] | Try before you buy; performance benchmarking; open Application Program Interface (API); preferred package and vendor lists; simplify glue code development |
| 3 | Component-based development paradigm[11] | Open API; agent-based testing; use cases to group components into test sets; test harness (with standard instrumentation to test fine-grained passive/active parts) |
| 4 | Java (active applets)* | Agent-based testing; fuzzy set theory (localization). Neural networks (dynamic instantiation); Java virtual machine restriction and instrumentation |
| 5 | Active agents (including web-based robots, spiders, etc.)* | Brute force testing using distributed test technology; Java testing concepts (see 4); knowledge-based extensions (smart agents; ORB based guardians, etc.) |

*Still being researched. Body of test knowledge about what works and what doesn't is not available yet for test community's use.

## Testing Software: Challenges for the Future

*Continued from page 3*

When you perform a root cause analysis to determine the real problem, you can make the following three observations:

- We never seem to allocate enough time and effort to testing activities. Even those who do seem to get into trouble because their management tends to reallocate these resources to others as problems arise. In response, maybe we should calibrate our estimation models more precisely to our actual test experience so that we have adequate resources when we start off. Then, we could put processes in place to allocate reserves retained to deal with risk instead of taking funds away from the testing effort.
- Management at all levels of the organization doesn't seem to fully understand what it takes to be successful in a test effort. They don't realize that large investment for processes, tools, techniques, facilities, and infrastructure is needed in order to put test technology to work for them. Perhaps, the test community needs to do a better job of educating their management about their needs. But, they need to do so armed with the data that was derived above about what it really takes to get the job done.
- Test management tends to pay more attention to the technical than the management issues. For example, they focus on test methods and tools instead of processes and infrastructure. I believe that the order of concentration should be reversed. Weave your test expectations into your standard software development process. If you use the Capability Maturity Model as your framework[12], do this in such a way that a test discipline is a natural part of the way that you conduct your business. If you are pursuing ISO certification, define quantitative test expectations for your gate checklists.

I'd like to issue a call to action. Let's do something about this state of affairs. Let's view these three observations as opportunities, not challenges. Instead of complaining that we don't have the resources to do the job right, let's gather data about our test experience and publish it, to help set reasonable expectations. Let's ask professional groups like the International Society of Parametric Analysts (ISPA) and the International Function Point Users Group (IFPUG) to prepare benchmarks about testing for the community. Let' stimulate more work on test issues within the universities and research institutions. Most importantly, let's use the data we publish to prepare business cases for improving our processes and inserting a viable test management infrastructure as we initiate our education activities.

### About the Author

Donald J. Reifer is one of the leading figures in the field of software engineering and management with over 30 years of progressive experience in both industry and government. Recently, Mr. Reifer managed the DoD Software Initiatives Office under an Intergovernmental Personnel Act assignment with the Defense Information Systems Agency (DISA). As part of this assignment, he also served as the Director of the DoD Software Reuse Initiative and Chief of the Ada Joint Program Office. Previously, while with TRW, Mr. Reifer served as Deputy Program Manager for their Global Positioning Satellite (GPS) efforts. While with the Aerospace Corporation, Mr. Reifer managed all of the software efforts related to the Space Transportation System (Space Shuttle).

Currently, as President of RCI, Mr. Reifer supports executives in many Fortune 500 firms who are looking to develop investment strategies and improve their software capabilities and capacity. Mr. Reifer is the Principal Investigator on our best software acquisition practices and information warfare SBIR efforts. He is also helping develop a variety of estimating models as a senior research associate on the USC COCOMO II team led by Dr. Barry Boehm.

Mr. Reifer was awarded the Secretary of Defense's Medal for Outstanding Public Service in 1995 for the innovations he brought to the DoD during his assignment. Some of his many other honors include the Hughes Aircraft Company Fellowship, the Frieman Award for advancing the field of parametrics, the NASA Exceptional Service Medal and membership in Who's Who in the West.

### Author Contact Information

**Donald J. Reifer**
Reifer Consultants Inc.
P.O. Box 4046
Torrance, CA 90510-4046
Phone: (310) 530-4493
Fax: (310) 530-4297
info@reifer.com
www.reifer.com

*References on page 5*

## References

**[1]** Edward Yourdon, *Death March*, Prentice-Hall, 1997.

**[2]** Publications like *Software Testing & Quality Engineering* and *Component Strategies*.

**[3]** Frederick P. Brooks, Jr., *The Mythical Man-Month*, Addison-Wesley, 1975.

**[4]** Tom DeMarco, *Controlling Software Projects*, Yourdon Press, 1982.

**[5]** Robert L. Glass, *In the Beginning: Recollections of Software Pioneers*, IEEE Computer Society, 1997.

**[6]** Robert Walker Royce, *Software Project Management: A Unified Framework*, Addison-Wesley, 1998.

**[7]** Hans-Erik Eriksson and Magnus Penker, *UML Toolkit*, John Wiley & Sons, 1998.

**[8]** Lawrence H. Putnam and Ware Myers, *Measures for Excellence*, Prentice-Hall, 1992.

**[9]** Software Productivity Consortium, *Software Measurement Guidebook*, International Thomson Computer Press, 1995.

**[10]** Philippe Kruchten, *The Rational Unified Process*, Addison-Wesley, 1999.

**[11]** Donald J. Reifer, *Product Line Management: Best Acquisition Processes/Practices Technology Transfer Kit*, Reifer Consultants, Inc., 1999.

**[12]** Mark C. Paulk, Charles V. Weber, Bill Curtis and Mary Beth Chrises, *The Capability Maturity Model: Guidelines for Improving the Software Process*, Addison-Wesley Publishing Co., New York, NY, 1995.

## Object-Oriented Technology (OOT) in Testing

***by Larry Bernstein, Have Laptop - Will Travel***

### Introduction

The telephone gadget is an enormously successful invention; each new level of system that surrounds it has spawned radical innovations and new services. Each changes adds complexity to managing the telephone network. Object-Oriented Technology (OOT) is the best answer to controlling the multiplying configurations that suddenly appear with new services. But OOT experiences the same birth pangs as every other new idea.

In addition, outsourcing is changing the telecommunications industry. In the last decade service providers have moved from developing their own services and equipment internally to buying them from third parties. The split of Lucent Technologies from AT&T in 1996 was the ultimate expression of this policy. With outsourcing comes the challenge of evaluating just how well vendor systems work before making a purchase decision.

### Test Models

GlobalOne met this challenge with an innovative use of Teradyne's TestMaster™ tool. TestMaster is an automated test design and coding tool, which was used for building an object oriented model of the outsourced system. The model was based on the specifications contained in their Request for Proposal and from system descriptions provided by the supplier. GlobalOne engineers were able to use this model to first map the functions they wanted against the system description and then against the system itself. This assured them that the contracted telephony functions were present and the GlobalOne system engineers understood how the new service would fit into their business environment. This approach showed how giving modeling tools to the customer system engineers can head off unintended consequences well before the system is even developed.

The TestMaster model of the service node gave the GlobalOne system engineers insight into the dynamics of this complex system of systems that made up the service offering. With the model, the systems engineers were able to study the unique call flow for every variation of the service. For example, GlobalOne customers can use one of 12 languages to interact with their network. Manual evaluation of the interaction of language selection based on the object libraries with the many service variations would have been a huge task without TestMaster and supporting evaluation tools. In traditional manual methods the system engineers would study the system specifications and then develop test cases to verify that the system worked as they expected. Finding the error paths is always a challenge. Typically many review meetings are needed among the system engineers themselves and

*Continued on page 6*

*Continued from page 5*

then with the vendor's technical people to ferret out the potential incompatibilities. With this approach, serious problems are often overlooked, which at best show up in service testing and at worst are found by the paying customers. Problems found and fixed during the service test effort cost three times the effort of those found with the model. Those found by the user add another factor of ten in cost escalation.

The TestMaster model-based test creation method permits the early involvement of the test organization in the development process, and is a powerful tool for facilitating communication between customer and supplier engineers. For example, the service offering systems use several different database technologies. To install the new service a database of customers was needed which contained administrative data and their service requests. The database initialization process was modeled with TestMaster, such that the database records were automatically generated from the model. Once the testers saw the strength of the model they adopted it as their test case database repository. Consequently, the TestMaster model of the databases was used for both populating the component databases in the target system, as well as serving as the input data for the test creation process. Expected results from the model were kept and later compared to the results from running the test cases against the system. When there were differences, analysts would compare the flow in the model with the flow in the service offering and find the problem. This moved debugging from detective work to analysis. Problems were found in the object libraries, component systems, in the model and even in the system design.

The model assures all features are present, not just the headliners. Once the service offering is installed in the evaluation labs the model produces test suites for automatic test drivers. These tests verify that the system performs as expected.

The test scripts from the model resulted in high coverage rates for feature testing. Quite often testers are pressed for time and do not have the resources for exhaustive load testing and reliability testing. While testers focus on full load testing they often do not have the time to run 'no-load' tests. These tests set up one or two simple transactions and then let the system idle waiting for new work. With the TestMaster model, setting up such a script was easy to do and pointed to reliability problems in the service offering system. With the data in hand it was clear that the offered load was triggering reliability problems and there was no argument that it was an unrealistic test load. A long-term benefit is that once the system is installed the model may be used for regression testing

## How to Reap Benefits

There are four broad benefits to using OOT: the technique can manage complexity better than anything else available; development and testing speed is increased; reuse becomes possible in a realistic way that has not been practical before; and it permits the scaling-up of systems. To reap these benefits, there are three factors that are critical: software testing techniques must be generally understood; tools and an infrastructure must be in place and middle management must accept the new mind-set so the culture and management processes are appropriate to this new method.

Most of what we hope to gain from OOT derives from module encapsulation, and the idea of pre-built libraries of modules or classes that are designed and tested for reuse.

I spoke with Bjarne Stroustrup, inventor of C++, and he remarked that OOT forces one to think about architecture and design from the beginning, not just as an after thought. Stroustrup, in his soft-spoken but direct way, points out that every significant change carries risk:

> *"My number one reason to worry about OOT is that middle managers will have to learn new tricks and techniques to manage development well. This is not easy. A development manager, say, is often in a position to get blamed for whatever goes wrong, yet most of the credit goes to their people when things go well. This doesn't encourage risk-taking or innovation. Unless managers somehow find the time to get a basic understanding and a minimum level of comfort with OOT, it will seem only a risk, and token efforts will abound."*

In 1988, I ran a project called MACSTAR whose purpose was to control changes on a Centrex system so people could have their telephone numbers follow them through moves. OOT was impressive here. There was a three to one increase in productivity. Testing proceeded quickly and new features were added to the object-oriented sections with less effort and with fewer interface changes within the system. Accommodations to external changes in the object-oriented modules caused significantly fewer source lines to be modified than for similar changes in structured design modules. Redesign of selected subsystems using the object-oriented paradigm produced savings in maintenance. This was, however, a small project, just 10K lines of code, 400 function points and 20 people.

Two years later when there was a major effort to deploy OOT throughout the organization, the problems inherent in larger systems appeared. There was legacy code to deal with, the tools were primitive, the mind set of relational databases was difficult to change, and there was a tendency to loose track of large numbers of object classes. There was some stumbling and hesitation. Testers became confused by the huge object libraries.

But the technology has improved enough to allow us to produce a system that changes the way of running a telephone network. Previously the focus had been on the various hardware architectures that would support broadband services. An integrated object oriented operations support system platform that enables the efficient delivery, differentiation and billing of new high-quality services became possible. Key was the ability to understand and test new services.

This project changed the paradigm of software development to such a degree that the projections of 3,000 people and 36 months to delivery were invalidated. It took 425 people 15 months to deliver the first release. There are 14 separate subsystems. Of these, eight are new designs and six are encapsulations of previously built systems which were not redone. There are 9K function points, 22 modules, 47 system interfaces and 12 databases. The object classes are of three types: communications; administration of the system itself; and user interfaces. There are 195 object classes and 376 objects. The communications object classes constituted 190K lines in a 3.5 million line system. The discipline required to do OOT exposed a 40% redundancy in the design specifications and allowed a paring down to elegance.

A guiding principle from our previous troubles was that the object class libraries should be very small, 0.5% of the number of function points, because we had no tools to keep track of large numbers of classes. There is a tendency for programmers to revert to the “cottage industry syndrome” of developing unique dialects that became unmaintainable. Testers were charged with accepting or rejecting new object libraries The details of the object classes caused problems at boundary conditions. For example, when the attributes of data being passed were slightly different, modules were not initialized properly. There was considerable churn on attributes. Ultimately Fred Brooks’ realization of the major paradigm shift was made real to us also: the greater the isolation of each object, the greater its power. Testers made sure that the isolation became a reality.

## About the Author

Mr. Bernstein is president of the Center of National Software Studies and is a recognized expert in Software Technology. He provides consulting through his firm Have Laptop - Will Travel and is the Executive Technologist with Network Programs, Inc. building software systems for managing telephone services.

Mr. Bernstein was an Executive Director of AT&T Bell Laboratories where he worked for 35 years.

## Author Contact Information

**Larry Bernstein**
Have Laptop-Will Travel
4 Marion Ave.
Short Hills, NJ 07078-2120
(973) 258-9213

lberstein@worldnet.att.net

# More Reliable, Faster, Cheaper Testing with Software Reliability Engineering

*by John D. Musa, Software Reliability Engineering and Testing Courses*

## Introduction

The testing of software systems is subject to strong conflicting forces. A system must function sufficiently reliably for its application, but it must also reach the market no later than its competitors (preferably before) and at a competitive cost. Government systems may be less market-driven, but balancing reliability, time of delivery, and cost is also important for them. One of the most effective ways to do this is to apply software reliability engineering to testing (and development)[1,2].

Software reliability engineering has resulted in a new view of testing, in which:

1. The most efficient testing involves activities throughout the entire life cycle;
2. Testers are empowered to take leadership positions in pro-actively meeting user needs; and
3. Testers collaborate closely with system engineers, system architects, users, managers, and developers.

Software reliability engineering delivers the desired functionality for a product much more efficiently by quantitatively characterizing its expected use. It uses this information to precisely focus resources on the most used and/or most critical functions (by "critical" I mean having great extra value when successful or great extra impact when failing with respect to human life, cost, or capability) and to have the tests realistically represent field conditions. Thus, software reliability engineering tends to increase reliability while decreasing development time and cost. Then software reliability engineering balances customer needs for the major quality characteristics of reliability, availability, delivery time, and life cycle cost more effectively by:

1. Setting quantitative reliability as well as schedule and cost objectives;
2. Engineering strategies to meet the objectives; and
3. Tracking reliability in test as a release criterion.

SRE is a proven, standard, widespread best practice that is built on a sound theoretical foundation[3] and is widely applicable. As an example, Tierney[4] reported the results of a survey taken in late 1997 that showed that Microsoft has applied software reliability engineering in 50 percent of its software development groups, including projects such as Windows NT and Word. It has been an AT&T best current practice since May 1991. Qualification as an AT&T best current practice requires widespread use, a documented large benefit/cost ratio, and a probing review by two boards of high-level managers. Some 70 project managers also reviewed the practice of software reliability engineering. Standards for approval as an AT&T best current practice are high; only five of 30 proposed best current practices were approved in 1991. An AIAA standard for software reliability engineering was approved in 1993, and IEEE standards are under development. McGraw-Hill and the IEEE Computer Society Press recently recognized the rapid maturing and standardization of the field, and have published a handbook on the topic[5].

SRE is low in cost and its deployment has very little schedule impact. You can apply software reliability engineering to any software-based system, including legacy systems, beginning at the start of any release cycle. It encourages greater communication among different project roles. With SRE, testers typically participate as members of the system engineering team. They help develop operational profiles, set failure intensity objectives, and select project reliability strategies.

SRE is very customer-oriented. It involves direct interaction with customers, and this enhances your image as a supplier, (if you have any reasonable degree of competence) improving customer satisfaction. SRE is highly correlated with attaining Levels 4 and 5 of the SEI Capability Maturity Model.

## Process Overview

Applying software reliability engineering to test involves five major activities: defining the "just right" reliability, developing operational profiles, preparing for test, executing test, and guiding test.

I will illustrate these activities in the context of an actual project at AT&T, which I call Fone Follower. I selected this example because of its simplicity; it in no way implies that software reliability engineering is limited to telecommunications systems. I have changed certain information to keep explanation simple and protect proprietary data.

Fone Follower is a system that lets telephone calls "follow" subscribers anywhere in the world (even to cell phones). Subscribers dial into a voice-response system and enter the telephone numbers at which they plan to be at various times. Incoming calls (voice or fax) that would normally be routed to a subscriber's telephone are then sent to Fone Follower, which forwards them in accordance with the program entered. If there is no response to a voice call and the subscriber has pager service, Fone Follower pages. If there is still no response or if the subscriber does not have pager service, Fone Follower forwards calls to the subscriber's voice mail.

## Defining "Just Right" Reliability

To define the "just right" level of reliability for the product, you set the failure intensity objective (FIO), balancing among major quality characteristics users need. A *failure* is a departure of system behavior in execution from user needs. *Failure intensity* is simply the number of failures per unit time. The best way to determine the FIO is to use field data from a similar release or product. This data includes customer satisfaction surveys related to measured failure intensity, and an analysis of competing products. Then you engineer project software reliability strategies to meet these objectives. For example, you may determine the resources you will devote to requirements reviews, the amount of unit test, the degree to which you will implement fault tolerant features, etc.

## Developing Operational Profiles

An operation is a major task of short duration performed by a system, which returns control to the system when complete. It is a logical rather than a physical concept, in that an operation can be executed over several machines and it can be executed in noncontiguous time segments. An operation can be initiated by a user, another system, or the system's own controller. Some examples of operations are a command activated by a user, a transaction sent for processing from another system, a response to an event occurring in an external system, and a routine housekeeping task activated by your own system controller. The operational profile is simply the set of operations and their probabilities of occurrence. An operational profile is a complete set of functions with their probabilities of occurrence. Table 1 shows an illustration of an operational profile from Fone Follower.

You can use operational profiles in system engineering to reduce the number of operations to those that are cost effective with respect to life cycle system costs and benefits, to plan a competitive release strategy (schedule a small number of most-used operations for a speeded-up first version and defer the others to a later version), and to focus resources on the functions and modules that are most used or most critical. But operational profiles will also play a major role in preparing for and executing a test.

To develop an operational profile, you identify the initiators of operations, enumerate the operations that are produced by each initiator, determine the occurrence rates of the operations, and determine the occurrence probabilities by dividing the occurrence rates by total operation occurrence rates.

Many first-time users of SRE think that determining operation occurrence rates will be very

**Table 1. Fone Follower Operational Profile**

| Operation | Occurrence Probability |
|---|---|
| Process voice call, no pager, answered | 0.18 |
| Process voice call, no pager, no answer | 0.17 |
| Process voice call, pager, answered | 0.17 |
| Process fax call | 0.15 |
| Process voice call, pager, answer on page | 0.12 |
| Process voice call, pager, no answer on page | 0.10 |
| Enter forwardees | 0.10 |
| Audit section - phone number database | 0.009 |
| Add subscriber | 0.0005 |
| Delete subscriber | 0.0005 |
| Recover from hardware failure | 0.000001 |
| **Total** | **1** |

*Continued on page 10*

## Testing with Software Reliability Engineering

*Continued from page 9*

difficult; our experience indicates much less difficulty than expected. Frequently, field data already exists for the same or similar systems, perhaps from previous versions. If not, you can often collect it. Even if there is no direct data, you can usually make reasonable estimates from related information. Finally, failure intensity achieved in test is very robust with respect to errors in operation occurrence rates.

### Preparing for Test

To prepare for test, we prepare the test cases and the test procedures. We allocate test cases to operations in accordance with their occurrence probabilities, with special consideration given to critical operations. We then select test cases within the operation on a uniform basis. Test procedures are load test controllers that set up environmental conditions and randomly select and invoke test cases from the test case set, based on the operational profile.

### Executing Test

We allocate test time among feature test, load test, and regression test. In feature test, test runs are executed essentially independently of each other, with interactions minimized. In load test, large members of test runs are executed simultaneously. Load test stimulates failures that can occur as a result of interactions among runs. In regression test, feature test runs are repeated after each build to see if any changes made to the system have spawned faults that cause failures. We identify failures, determine when they occurred, and establish the severity of their impact.

### Guiding Test

You will interpret failure data differently for software you are developing and software you are acquiring. For software you are developing you attempt to remove the faults that are causing failures. You track progress, generally at fixed time intervals, by looking at the failure intensity to failure intensity objective (FI/FIO) ratio. For software you are acquiring (this can be by contract, purchase, or reuse from a library), you determine whether that software should be accepted or rejected, with limits on the risks taken. For acquired software, you interpret failure data after each failure.

For developed software, we estimate the FI/FIO ratio from the times of failure events or the number of failures per time interval, using reliability estimation programs such as CASRE[5]. These programs are based on software reliability models and statistical inference. Figure 1 shows a typical plot of the FI/FIO ratio. Significant upward trends in the plot commonly indicate nonstationary test selection or system evolution due to poor change control. Both need correction if you are to have a quality test effort that you can rely on. We consider releasing the software when the FI/FIO ratio reaches 0.5.

For acquired software we apply a reliability demonstration chart, shown in Figure 2. Failure times are normalized by multiplying by the failure intensity objective. Each failure is plotted on the chart. Depending on the region in which it falls, you may accept or reject the software being tested or continue to test. Figure 2 shows a test in which the first two failures indicate you should continue testing, and the third failure recommends that you accept the software.

Charts can be constructed for different levels of consumer risk (the risk of accepting a bad program) and supplier risk (the risk of rejecting a good program).

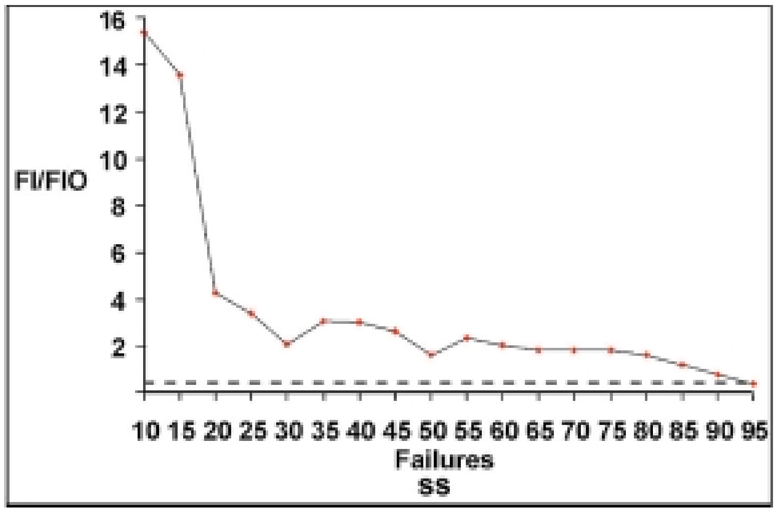


**Figure 1. Plot of FI/FIO Ratio**

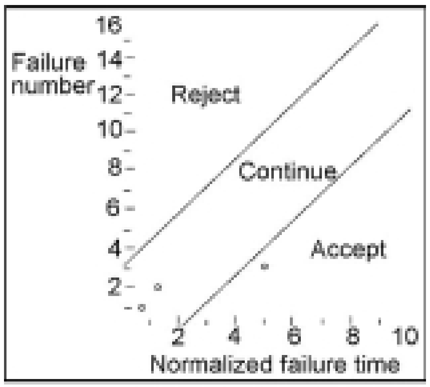


**Figure 2. Reliability Demonstration Chart**

## Conclusion

Practitioners in many organizations (see[1,2] for lists) have found software reliability engineering unique in providing a standard proven way to engineer testing for confidence in the reliability of the software-based systems they deliver, as they deliver them in minimum time with maximum efficiency. It is a vital skill in today's marketplace.

## About the Author

John D. Musa teaches courses and consults in software reliability engineering and testing. He has been involved in software reliability engineering since 1973 and is generally recognized as one of the creators of that field. Recently, he was Technical Manager of Software Reliability Engineering at AT&T Bell Laboratories, Murray Hill. He organized and led the transfer of software reliability engineering into practice within AT&T, spearheading the effort that defined it as a "best current practice." Musa has also been actively involved in research to advance the theory and practice of software reliability engineering. He has published more than 100 articles and papers, given more than 175 major presentations, and made several videos. He is principal author of *Software Reliability: Measurement, Prediction, Application* and author of *Software Reliability Engineering: More Reliable Software, Faster Development and Testing*. Musa received an MS in electrical engineering from Dartmouth College. He has been listed in Who's Who in America and American Men and Women of Science since 1990. He is a fellow of the IEEE and the IEEE Computer and Reliability Societies and a member of the ACM and ACM Sigsoft.

## Author Contact Information

**John D. Musa**
39 Hamilton Road
Morristown, NJ 07960-5341

(973) 267-5284
Fax: (973) 267-6788
j.musa@ieee.org
http://members.aol.com/JohnDMusa

## References

**[1]** Musa, J. D. *Software Reliability Engineering* Website: overview, briefing for managers, bibliography of articles by software reliability engineering users, information on courses, useful references, Question of the Month: http://members.aol.com/JohnDMusa/

**[2]** Musa, J. D. *Software Reliability Engineering: More Reliable Software, Faster Development and Testing*, ISBN 0-07-913271-5, McGraw-Hill, New York, 1998.

**[3]** Musa, J.D., A. Iannino, and K. Okumoto. *Software Reliability: Measurement, Prediction, Application,* ISBN 0-07-044093-X, McGraw-Hill, New York, 1987.

**[4]** Tierney, J. *SRE at Microsoft*. Keynote speech at the 8th International Symposium on Software Reliability Engineering, Albuquerque, NM. 1997.

**[5]** Lyu, M. (Editor). *Handbook of Software Reliability Engineering*, ISBN 0-07-039400-8, McGraw-Hill, New York (includes CD/ROM of CASRE program). 1996.

# Website Testing

***by Edward Miller, Software Research Inc.***

## Introduction

The nearly instant worldwide audience makes a Website's quality and reliability crucial to its success. The nature of the WWW and of Website software pose unique software testing challenges. Webmasters, WWW applications developers, and Website quality assurance managers need tools and methods that meet very specific needs. Our technical approach, based on extending existing WWW browsers, offers many attractive benefits in meeting these needs.

## Background

Within minutes of going live, a WWW application can have many thousands more users than a conventional, non-WWW application. The immediacy of a Website creates immediate expectations of quality, but the technical complexities of a Website and variances in the available browsers make testing and quality control more difficult than "conventional" client/server or application testing. Automated testing of Websites thus is both an opportunity and a significant challenge.

## Defining Website Quality & Reliability

Like any complex piece of software there is no single quality measure to fully characterize a Website. There are many dimensions of quality, and each measure will pertain to a particular Website in varying degrees. Here are some of them:

- *Timeliness*: How much has a Website changed since the last upgrade?
- *Structural Quality*: Are all links inside and outside the Website working? Do all of the images work?
- *Content*: Does the content of critical pages match what is expected?
- *Accuracy and Consistency*: Are today's copies of the pages downloaded the same as yesterday's?
- *Response Time and Latency*: Does the Website server respond to a browser request within certain parameters? In an E-commerce context, what is the end to end response time after a SUBMIT?
- *Performance*: Is the Browser -> Web -> Website -> Web -> Browser connection quick enough? How does the performance vary by time of day, by load and usage?

Clearly, "Quality" is in the mind of the Website user. A poor-quality Website, one with many broken pages and faulty images, with CGI-bin error messages, etc. may cost in poor customer relations, lost corporate image, and even in lost sales revenue. Very complex Websites can sometimes overload the user.

## Website Architectural Factors

A Website can be quite complex, and that complexity can be a real impediment in assuring Website quality.

What makes a Website complex? These are the issues test systems have to contend with:

**Browser**. There is a kind of de facto standard: the Website must use only those constructs that work with the majority of browsers. But this still leaves room for a lot of creativity, and a range of technical difficulties.

**Display Technologies**. What you see in your browser is actually composed from many sources: (1) HTML: Various versions of HTML must be supported.(2) Java, JavaScript, ActiveX: Obviously JavaScript and Java applets are likely parts of a Website, and the quality process must support these. (3) CGI-Bin Scripts: All of the different types of CGI-bin scripts (Perl, awk, shell-scripts, etc.) need to be handled; tests will need to check "end to end" operation.

**Navigation**. Navigation in a Website often is complex and has to be quick and error free.

**Object Mode**. The display you see in a browser changes dynamically; the only constants are the "objects" that make up the display. Testing ought to be in terms of these objects.

**Server Response**. How fast the Website host responds influences whether a user moves, continues, or gives up.

## Website Test Automation Requirements

Assuring Website quality automatically requires conducting sets of tests, automatically and repeatably, that demonstrate required properties and behaviors. Here are some required elements of tools that aim to do this.

**Browser Independent**. Tests should be realistic, but not be dependent on a particular browser.

**No Buffering, Caching**. Local caching and buffering should be disabled so that timed experiments are a true measures of performance.

**Object Mode**. Object mode operation is essential to protect an investment in test suites and to assure that test suites continue operating when Website pages experience change.

**Tables and Forms**. Even when the layout of a table or form varies in the browser's view, tests of it should continue independently of these factors.

Tests need to operate from the browser level for two reasons: this is where users see a Website, so tests based in browser operation are the most realistic; and tests based in browsers can be run locally or across the Web equally well. Local execution is fine for quality control, but not for performance measurement work, where response time including Web-variable delays reflective of real-world usage is essential.

## Website Dynamic Validation

Confirming the validity of what is tested is the key to assuring Website quality, the most difficult challenge of all. Here are four key areas where test automation will have a significant impact.

**1. Operational Testing.** Individual test steps may involve a variety of checks on individual pages in the Website:

- *Page Consistency*. Is the entire page identical to a prior version? Are key parts of the text the same or different?
- *Table, Form Consistency*. Are all of the parts of a table or form present? Correctly laid out? Can you confirm that selected texts are in the "right place"?
- *Page Relationships*. Are all of the links on a page the same as they were before? Are there new or missing links? Are there any broken links?
- *Performance Consistency, Response Times*. Is the response time for a user action the same as it was (within a range)?

**2. Test Suites.** Typically you may have dozens or hundreds (or thousands?) of tests, and you may wish to run these tests in a variety of modes: unattended, distributed across many machines, background, etc.

**3. Content Validation.** Apart from how a Website responds dynamically, the content should be checkable either exactly or approximately. Here are some ways that content validation could be accomplished:

**Structural**. All of the links and anchors should match with prior "baseline" data.

**Checkpoints**, **Exact Reproduction**. One or more text elements in a page should be markable as "required to match".

**Selected Images/Fragments**. The tester should be able to rubber-band sections of an image and require that the selection image match later during a subsequent rendition of it.

**4. Load Simulation.** Load analysis needs to proceed by having a special purpose browser act like a human user. This assures that the performance checking experiment indicates true performance - not performance on simulated but unrealistic conditions. There are many "http torture machines" that generate large numbers of http requests, but that is not necessarily the way real-world users generate requests.

## Testing System Characteristics

Considering all of these disparate requirements, it seems evident that a single product that supports all of these goals will not be possible. However, there is one common theme: the majority of the work seems to be based on "What does the Website look like from the point of view of the user?" That is, from the point of view of someone using a browser to look at the Website.

This observation led our group to conclude that it would be worthwhile trying to build certain test features into a "test enabled web browser", which we called CAPBAK/Web in the expectation that this approach would let us do the majority of the Website quality control functions using that engine as a base.

**Browser Based Solution.** With this as a starting point, we determined that the browser based solution had to meet these additional requirements:

- *Commonly Available Technology Base*. The browser had to be based on a well known base (there appear to be only two or three choices).

*Continued on page 14*

*Continued from page 13*

- *Some Browser Features Must Be Deletable*. At the same time, certain requirements imposed limitations on what was to be built. For example, if we were going to have accurate timing data we had to be able to disable caching because otherwise we are measuring response times within the client machine rather than "across the Web."
- *Extensibility Assured*. To permit meaningful experiments, the product had to be extensible enough to permit timings, static analysis, and other information to be extracted.

Taking these requirements into account, and after investigation of W3C's Amaya Browser and the open-architecture Mozilla/Netscape Browser we chose the IE Browser as our initial base for our implementation of CAPBAK/Web.

**User Interface**. How the user interacts with the product is very important, in part because in some cases the user will be someone very familiar with Website browsing and not necessarily a testing expert. The design we implemented takes this reality into account.

**"C" Scripting**. We use interpreted "C" language as the control language because the syntax is well known, the language is fully expressive of most of the needed logic, and because it interfaces well with other products.

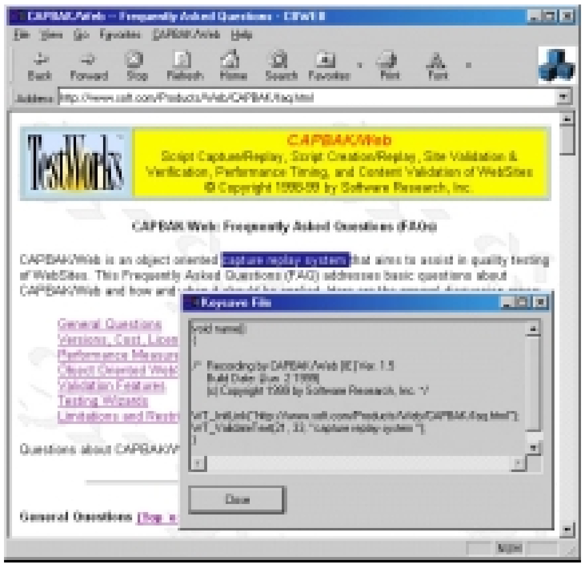
**Files Interface**. We implemented a set of dialogs to capture critical information and made each of them recordable in a text file. The dialogs are associated with files that are kept in parallel with each browser invocation:

- *Keysave File*. This is the file that is being created -- the file is shown line by line during script recording as the user moves around the candidate Website.
- *Timing File*. Results of timings are shown and saved in this file.
- *Messages File*. Any error messages encountered are delivered to this file. For example, if a file can not be downloaded within the user-specified maximum time an error message is issued and the playback continues. (This helps preserve the utility of tests that are partially unsuccessful.)
- *Event File*. This file contains a complete log of recording and playback activities that are useful primarily to debug a test recording session or to better understand what actually went on during playback.

**Operational Features.** Based on prior experience, the user interface for CAPBAK/Web had to provide for several kinds of capabilities already known to be critical for a testing system. Many of these are critically important for automated testing because they assure an optimal combination of test script reliability and robustness.

- *Script Capture/Replay*. We had to be able to capture a user's actual behavior on-line, and be able to create scripts by hand.
- *Object Mode*. The recording and playback had to support pure-Object Mode operation. This was achieved by using internal information structures in a way that lets the scripts (either recorded or constructed) refer to objects that are meaningful in the browser context.
- [*Adjustable*] *True-Time Mode*. We assured realistic behavior of the product by providing for recording of user-delays and for efficient handling of delays by incorporating a continuously variable "playback delay multiplier" that can be set by the user.
- *Playback Synchronization*. For tests to be robust there must be a built-in mode that assures synchronization so that Web-dependent delays do not interfere with proper Website checking. CAPBAK/Web does this using a proprietary playback synchronization method that waits for download completion (except if a specified maximum wait time is exceeded).
- *Timer Capability*. To make accurate on-line performance checks we built in a 1 millisecond resolution timer that could be read and reset from the playback script.
- *Validate Selected Text Capability*. A key need for Website content checking, as described above, is the ability to capture an element of text from an image so that it can be compared with a baseline value. This feature was implemented by digging into the browser data structures in a novel way (see below for an illustration). The user highlights a selected passage of the Web page and clicks on the "Validate Selected Text" menu item.

  What results is a recorded line that includes the ASCII text of what was selected, plus some other information that locates the text fragment in the page. During


CAPBAK/Web -- Frequently Asked Questions - CBWT 8
File View Go Favorites CAPBAK/Web Help
Back Forward Stop Refresh Home Search Favorites Print Font
Address http://www.soft.com/Products/Web/CAPBAK/faq.html
TestWorks
CAPBAK/Web
Script Capture/Replay, Script Creation/Replay, Site Validation & Verification, Performance Timing, and Content Validation of WebSites
© Copyright 1998-99 by Software Research, Inc.
CAPBAK Web: Frequently Asked Questions (FAQs)
General Questions
Versions, Cost, Licen
Performance Measure
Object Oriented Web
Validation Features
Testing Wizards
Limitations and Restri
Questions about CAPBAK/W
General Questions [Top o
Keysave File
Close

CAPBAK/Web is an object oriented capture replay system that aims to assist in quality testing of WebSites. This Frequently Asked Questions (FAQ) addresses basic questions about CAPBAK/Web and how and

```
void name()
{

/* Recording by CAPBAK/Web [IE] Ver. 1.5
   Build Date: Sun. 2 1999
   (c) Copyright 1999 by Software Research, Inc. */

WT_InitLink("http://www.soft.com/Products/Web/CAPBAK/faq.html");
WT_ValidateText(21 , 33, "capture replay system ");
}
```

**Figure 1. Illustration of CAPBAK/Web Validate Selected Text Feature**

playback if the same text is not found at the same location an error message is generated.

- *Multiple-playback*. We confirmed that multiple playback was possible by running separate copies of the browser in parallel. This solved the problem of how to multiply a single test session into a number of test sessions to simulate multiple users each acting realistically.

**Test Wizards.** In most cases manual scripting is too laborious to use and making a recording to achieve a certain result is equally unacceptable. We built in several test wizards that mechanize some of the most common script-writing chores.

- *Link Wizard*. This wizard creates a script based on the current Web page that visits every link in the page. Scripts created this way are the basis for "link checking" test suites that confirm the presence (but not necessarily the content) of URLs.
- *FORM Wizard*. For E-Commerce testing which involves FORMS we included in the system a FORM Wizard that generates a script that:
  - o Initializes the form,
  - o Presses each push-button by name,
  - o Presses each radio button by name,
  - o Types a pre-set script fragment into each text field, and
  - o Presses SUBMIT.

The idea is that this script can be processed automatically to produce the result of varying combinations of pushing buttons. As is clear, the wizard will have pushed all buttons, but only the last-applied one in a set of radio buttons will be left in the TRUE state.

- *Text Wizard*. For detailed content validation the Text wizard yields up a script that includes confirmation of the entire text of the candidate page. This script is used to confirm that the content of a page has not changed (in effect, the entire text content of the subject is recorded in the script).

Figure 2 shows a sample of the output of theForm Wizard, applied to our standard test page.

## Example Uses

Early applications of the CAPBAK/Web system have been very effective in producing experiments and collecting data that is very useful for Website checking. While we expect CAPBAK/Web to be the main engine for a range of Website quality control and testing activities, we've chosen two of the most typical -- and most important -- applications to illustrate how CAPBAK/Web can be used.

**Performance Testing Illustration**. To illustrate how CAPBAK/Web measures timing we have built a set of Public Portal Performance Profile Test Suites that have these features:

- *Top 20 Web Portals*. We selected 20 commonly available Websites on which to measure response times. These are called the "P4" suites.
- *User Recording*. We recorded one user's excursion through these suites and saved that keysave file (playback script).
- *User Playback*. We played back the scripts on a 56 kbps modem so that we had a realistic

*Continued on page 16*

*Continued from page 15*

```
void name()
{
/* Produced by CAPBAK/Web [IE] Ver. 1.5 Form Wizard */
/* (c) Copyright 1999 by Software Research, Inc. */

WT_InitLink("http://www.testworks.com/Products/Web/CAPBAK/example1/");
WT_SubmitForm(FORM:0:12, "RESET FORM");
WT_SelectOneRadio(FORM:0:0, "now", "TRUE");
WT_SelectOneRadio(FORM:0:1, "next", "TRUE");
WT_SelectOneRadio(FORM:0:2, "look", "TRUE");
WT_SelectOneRadio(FORM:0:3, "no", "TRUE");
WT_SelectCheckBox(FORM:0:4, "concerned", "TRUE");
WT_SelectCheckBox(FORM:0:5, "info", "TRUE");
WT_SelectCheckBox(FORM:0:6, "evaluate", "TRUE");
WT_SelectCheckBox(FORM:0:7, "send", "TRUE");
WT_FormTextInput(FORM:0:8, "TestWorks");
WT_FormTextInput(FORM:0:9, "TestWorks");
WT_FormTextInput(FORM:0:10, "TestWorks");
WT_FormTextInput(FORM:0:11, "TestWorks");
WT_SubmitForm(FORM:0:13, "SUBMIT FORM");
}
```

**Figure 2. Sample of Output of FORM Test Wizard**

comparison of how long it would take to make this very-full visit to our selected 20 portals.

- *P4 Timings*. We measured the elapsed time it took for this script to execute at various times during the day. The results from one typical day's executions showed a playback time range of from 457 secs to 758 secs (i.e., from -19% of the average to +36% of the average playback time).
- *Second Layer Added*. We added to the base script a set of links to each page referenced on the same set of 20 Websites. This yielded the P4+ suite that visits some 1,573 separate pages, or around 78 per Website. The test suite takes around 20,764 secs (~5 Hrs 45 mins) to execute, or an average of 1,038 secs per Website.
- *Lessons Learned*. It is relatively easy to configure a sophisticated test script that visits many links in a realistic way, and provides realistic user-perceived timing data.

**E-Commerce Illustration.** This example shows a typical E-Commerce product ordering situation. The script automatically places an order and uses the Validate Selected Text sequence to confirm that the order was processed correctly. In a real-world example this is the equivalent of (i) selecting an item for the shopping basket, (ii) ordering it, and (iii) examining the confirmation page's order code to assure that the transaction was successful. (The final validation step of confirming that the ordered item was actually delivered to a specific address is not part of what CAPBAK/Web can do.)

- *Typical Order Form*. We based this script on a typical order form that collects customer information including, for example, a code number (a credit card number).
- *Type-In with Code Number*. Starting with the FORM Wizard generated script, we modify it to include only the parts we want, and include the code number 8889999 (See Figure 1).
- *Response File*. Once the playback presses the SUBMIT button the WebServer response page shows up displaying the response to the code number. We use the Validate Selected Text feature (see Figure 1) to capture the response number text.
- *Error Message Generated*. If the CGI-bin scripts make a mistake this will be caught during playback because the expected exact text 8889999 will not be present, it will be something else.
- *Completed Test* Script. Figure 3 is an illustration of the complete test script for CAPBAK/Web that illustrates this sequence of activities.
- *Lessons Learned*. This example illustrates how it is possible to automatically validate a Website using CAPBAK/Web by detecting when an artificial order is misprocessed.

```
void name()
{
/*  Recording by CAPBAK/Web [IE] Ver. 1.5  (c) Copyright 1999 by Software Research, Inc. */

WT_InitLink("http://www.soft.com/Products/Web/CAPBAK/example1/example1broken.html");
WT_SelectOneRadio(FORM:1:0, "buying-now", "TRUE");
WT_SelectOneRadio(FORM:1:1, "next-month", "FALSE");
WT_SelectOneRadio(FORM:1:2, "just-looking", "FALSE");
WT_SelectOneRadio(FORM:1:3, "no-interest", "FALSE");
WT_SelectOneRadio(FORM:1:4, "Yes", "TRUE");
WT_SelectOneRadio(FORM:1:5, "Yes", "TRUE");
WT_SelectOneRadio(FORM:1:6, "Yes", "TRUE");
WT_SelectOneRadio(FORM:1:7, "Yes", "TRUE");
WT_FormTextInput(FORM:1:8, "Mr. Software");
WT_FormTextInput(FORM:1:9, "415-957-1441");
WT_FormTextInput(FORM:1:10, "info@soft.com");
WT_FormTextInput(FORM:1:11, "8889999");
WT_SubmitForm(FORM:1:13, "SUBMIT FORM");
WT_Wait(3425);
WT_ValidateText(12, 143, "88899999");
}
```

**Figure 3.  Script for E-Commerce Test Loop**

## Summary

All of these needs and requirements impose constraints on the test automation tools used to confirm the quality and reliability of a Website. The CAPBAK/Web approach offers some significant benefits and technical advantages when dealing with complicated Websites.  Better, more reliable Websites should be the result.

## Resources

This paper is based on many sources and relies in part on a prior White Paper found at:  www.soft.com/Products/Web/Technology/website.quality.challenge.html

A more complete version of this paper can be found at: www.soft.com/Products/Web/Technology/website.testing.html

You can learn more about the CAPBAK/WEB system by taking a tour at:  www.soft.com/Products/Web/CAPBAK/Documentation.IE/CBWeb.GUI5.html

There is a very detailed description of the P4 Family of CAPBAK/Web examples at: www.soft.com/Products/Web/CAPBAK/pppp.html

## About the Author

Dr. Edward Miller is Chairman of Software Research, Inc., San Francisco, California, where he has been involved with software test tools development and software engineering quality questions. Dr. Miller has worked in the software quality management field for 25 years in a variety of capacities, and has been involved in the development of families of automated software and analysis support tools. He was chairman of the 1985 1st International Conference on Computer Workstations, and has participated in IEEE conference organizing activities for many years. He is the author of *Software Testing and Validation Techniques*, an IEEE Computer Society Press tutorial text. Dr. Miller received a Ph.D. (Electrical Engineering) degree from the University of Maryland, an M.S. (Applied Mathematics) degree from the University of Colorado, and a BSEE from Iowa State University.

## Author Contact Information

**Edward Miller**
Software Research, Inc.
901 Minnesota Street
San Francisco, CA 94107
(800) 942-SOFT
miller@soft.com
www.soft.com

# Improving Information Quality for the Warfighter through Self-Checking Systems

*by Tod Reinhart, Air Force Research Laboratory, Dr. D. Joel Mellema, Raytheon Systems Company, and Carolyn Boettcher, Raytheon Systems Company*

## Introduction

Testing of mission-critical systems to a high degree of reliability has been a long time problem for the military services. As a result, system failures may occur in the field due to faults that result from unusual environmental conditions or unexpected sequences of events that were never encountered in the laboratory. The types of systems we are interested in are those that must operate within the constraint of real-time deadlines and produce inexact outputs based on computations from a succession of heuristic and approximate algorithms. Such systems, which include complex software and hardware interactions, are particularly difficult to validate and test. To improve the validation and test process and deliver more reliable systems, we have been experimenting with self-checking systems that continuously monitor themselves to detect suspicious events, which may indicate residual errors at a deep performance level.

Under the Air Force Self Checking Embedded Information System Software (SCEISS) program, theoretical university results are being extended to new classes of problems while applying these new types of result checkers to real-time embedded applications. The preliminary results reported here include a description of the example applications and their checkers, the process used to select checkers, and some initial data points on any additional software costs that may accrue due to the development and test of checkers.

## Problem Statement

Like the military services, Raytheon has a long-term interest in and commitment to solving the problem of residual errors that are not detected and corrected before a system is deployed. Such errors are like a time bomb, waiting for just the right combination of circumstances to cause significant performance degradation or even catastrophic system failure. To find solutions to this problem, several years ago Raytheon surveyed problem reports of errors that were not detected in a production radar system until after radar subsystem integration. It was found that many of the errors resulted from an unusual combination of circumstances that were unlikely to be encountered in the integration laboratory and that might not even be encountered during operational testing, e.g., flight test.

Testing is the most widely used method of validating that systems are performing as requirements dictate. However, it has been shown that it is not feasible to use testing alone to validate large systems to a high degree of reliability[1]. As an alternative, correctness-proving techniques are sometimes used to verify system correctness. However, because of the difficulty of these techniques, it is not practical to apply them to anything but small, well contained portions of larger systems, such as a trusted kernel.

Another approach that is often used to ensure reliable performance is functional redundancy. For example, three versions of software may be independently implemented to perform a given function. The results of each version are compared, with a two out of three majority declared to be correct. However, there are a number of practical problems with functional redundancy. Often the implementations whose results are compared are not statistically independent, because even though system designers and implementers are working completely independently, they often make correlated errors. In addition, the amount of run-time resources required to execute the function is increased by a factor of three. In systems where run-time resources are tight, this multiplicative increase may not be acceptable.

How then can residual software errors, which significantly increase the DoD's cost of ownership and may contribute to mission failures, be significantly reduced or effectively eliminated? We concluded that an entirely new paradigm was needed to ensure that large, complex systems with limited run-time resources can be maintained in a cost effective manner so that they continue to operate correctly.

## Self-Checking Embedded Information System Technology

The Self Checking Enbedded Information System Software (SCEISS) program is applying Checker technology to a range of embedded information system applications with the expectation that this technology will result in a dramatic reduction in fielded software errors and maintenance costs. Checker technology is based on checker software that executes at critical points to check the correctness of intermediate system results. Checkers become a permanent part of the software - they are retained throughout the system operational life cycle. Checkers keep checking and checking and checking… even after the system is deployed. Because they are always checking, checkers will eventually encounter and detect those errors and unexpected conditions that cause degraded system performance whenever and wherever those conditions occur. Checkers can be considered to be analogous to hardware built-in-test that executes periodically throughout the mission to decide whether the hardware is functioning correctly. Like hardware built-in-test, checkers report any anomalies detected during the mission so that errors can be fixed.

## Definition of a Simple Checker

A simple checker is special software that is embedded in code to continually check results over a large number of executions. It must have a good probability of eventually detecting any errors, especially after many executions, while maintaining a very low probability of false alarm (i.e., a simple checker must rarely or never declare a correct result to be erroneous). By definition, a simple checker must be much simpler and more reliable than the algorithm being checked. As a result of this definition, the execution and memory overhead added by a checker is small. In addition, the fact that the checker is simpler than the original algorithm helps to ensure its statistical independence from the algorithm being checked**.** Although there are some similarities between checkers and traditional fault tolerance techniques based on redundancy, checkers are different from traditional software fault tolerance techniques in two important ways: they are statistically independent from the original algorithm; and they do not double or triple the cost and runtime overhead of a function being checked.

## Prior Checker Research

Universities, Raytheon, and the Air Force have sponsored prior checker research. The seminal research in checkers has been ongoing for more than 10 years, led by Dr. Manuel Blum at the University of California at Berkeley, who defined the results checking paradigm that is the foundation for the SCEISS effort[2]. After determining that theoretical checker results might be applicable to real-life embedded applications, Raytheon and the University of California jointly funded research to extend checkers to some realistic avionics applications. In particular, this effort resulted in the definition of a general method for checking the results of a Fourier transform implemented in limited precision, fixed point arithmetic, as reported in [3] and [4]. With Dr. Blum consulting, Raytheon began a small checker pilot program as part of a production radar upgrade program, described in [5]. We determined that two of the three checkers developed under the pilot program, but not deployed, would have detected errors that were uncovered later during flight test.

## What's Good About Results Checking?

Many undetected software errors involve rare combinations of circumstances. Because of their rarity, a very large number of independent tests would be needed to create these special circumstances, if they can be reproduced at all in the laboratory. Because they are embedded in the operational software, checkers can detect errors during all phases of testing and operational use. In fact, checkers can execute as often as the software executes. For example, a computation occurring every 10 milliseconds will be checked a million times every three hours. In addition, checkers can detect erroneous results that do not produce obvious symptoms at the system level, and thus, might easily be overlooked by testers. As a result, errors can be corrected even before they cause externally observable system degradation.

Checkers are independent of the system design methodology, the software implementation language, and the embedded processor on which the software executes. Instead, checkers are based on a priori mathematical principles or physical laws governing the computations being checked. As a result, checkers can be applied to legacy systems that are being upgraded almost as easily as they can be applied to new systems. Although particular checkers may be application domain dependent, the self-checking approach can be generally applied in any application domain.

*Continued on page 20*

## Self-Checking Systems

*Continued from page 19*

### Checking for the Pentium Division Bug

The division bug in Intel's original release of the Pentium processor provides an excellent example of the potential value of checkers and illustrates that checkers can be useful for detecting hardware bugs as well as software bugs. The Pentium division bug evidenced itself very rarely, less than 1 in every 8 billion inputs. However, even though the bug rarely caused a problem, users of the Pentium processor soon discovered it. As a result, Intel was forced to recall the "buggy" processors and correct the error.

When news of the Pentium division bug was published, Blum and Wasserman invented a software checker/corrector that provides an "a priori" solution to the Pentium bug[6]. Their checker not only detects the erroneous division result, but also corrects the result. The Pentium division checker/corrector does not even need to know the type of error or even that an error is present in order to detect and correct it.

### Applying Checkers to a Production Radar Upgrade Program

Begun early in 1998, SCEISS is demonstrating the efficacy of checkers when used as an integral part of the system development process to improve the quality of delivered systems[7]. SCEISS will also help to transition checkers from theory to practice. In 1998, SCEISS applied self-checking technology to a production radar upgrade program that was adding a range gated high (RGH) pulse repetition frequency (PRF) mode to the existing software. The mode code is reused from another radar program with substantial modifications needed to adapt it to a different platform. The SCEISS team analyzed the software requirements for the modifications and identified candidate functions for checking.

### Checking the CFAR Loop

From the candidate functions identified during the requirements analysis, the constant false alarm rate (CFAR) control loop was selected for checking. There were several reasons why the CFAR loop was considered a good candidate.

- The CFAR control loop has a direct impact on detected targets and false alarms that are critical performance measurements.
- Domain experts agreed that the CFAR loop often caused problems during radar system integration and flight test.
- The requirements were quite complicated, making it more likely that coding errors would occur in implementing them.
- The CFAR loop is a specific example of a general type of control loop calculation. Hence, a checker invented here might be applicable across a wide class of control problems.

### CFAR Checker Design

The CFAR threshold calculation is an example of an algorithm where there are no simple rules for determining if a result is right or wrong. As a result, our CFAR checker looks for "suspicious" results that probably indicate significant system performance degradation, rather than definitely "wrong" results.

In general control theory, the control loop seeks a stable threshold value that keeps the system operating in an optimal manner, even while the environment, as measured by the sensor being controlled, is changing. To do this, the threshold must be continually adapted to changing environmental conditions based on feedback from the sensor. The requirements in this case specify minimum and maximum values for the CFAR threshold. If the calculated value is outside of the specified range of 3.0 to 5.8, it is reset to the maximum or minimum value as appropriate.

To help quantify "suspicious" values, we first simulated the CFAR threshold calculation 100,000 times using randomly generated input data representative of that seen in a real system. Based on statistics of the expected distribution of threshold values, we predicted that the threshold value would be in the range of 3.5 to 3.8 for approximately 98% of the time. Therefore, values outside of the range 3.5 to 3.8 might be considered suspicious. We subsequently decided to set the checker to fire more conservatively at values outside of 90% of the legal range, or 3.28 - 5.52, respectively. In addition, the checker fired only when the calculated threshold exceeded the high or low value over 90% of the time after 50 values were collected at half-second intervals.

### Metrics Collected

The radar development team and SCEISS team worked together to implement the simple checker described above. In coding the checker, we added 15 Jovial source lines of code (SLOC) to the software module that calculated the CFAR loop threshold, which was originally about 100 SLOC. In addition, 19 SLOC were added to the Jovial Compool containing the global data definitions. The percentage increase in the SLOC gives a rough estimate of the percentage additional effort that would be required to implement checkers, assuming that the same productivity rate is used to predict the cost of developing checkers as is used to predict the cost of developing the application.

The performance overhead was estimated by comparing the operations performed by the checker compared to the operations performed by the algorithm. Performance overhead for the CFAR loop checker was estimated to be less than 5%. Checking was performed in the system integration laboratory in parallel with the regular system integration effort, so the program's flight test schedule would not be perturbed. The checker was run on four separate occasions in the integration laboratory against three versions of the software. On the first three occasions, the threshold values were consistently low causing the checker to fire under several different test conditions.

The software development team responsible for the production tape upgrade was informed about the problems with the CFAR loop uncovered by the checker. Subsequently, the application code was corrected and the checker was run for the fourth and final time in the integration laboratory after the system had been in flight test for some time. This final checker demonstration showed that the CFAR loop performance had been considerably improved.

We believe that this experiment illustrates the value of even very simple checkers. We found that even with a relatively insensitive checker, we were able to detect suspicious events occurring in the system which otherwise might not have been detected. For a more detailed description of the CFAR loop checker, the reader is referred to the SCEISS Interim Report for 1998[8].

## Applying Checkers to a Technology Program: Ongoing Proof of Concept Demonstration

In 1999, as our second proof-of-concept demonstration of checkers under SCEISS, we chose the Theatre Missile Defense Smart Sensor and Automatic Target Recognition (TESSA) program. TESSA is in the fourth phase of a technology demonstration using the F-15E aircraft and radar and FLIR sensor suite. During the TESSA IV flight test, fusion of SAR and IR sensor data for Automatic Target Cueing and Recognition in attack operations will be evaluated. In the demonstration, the SAR will be used to cue the IR sensor to interesting objects. The flight test will include modified APG-70 radar modes and a modified LANTIRN targeting pod, as well as new sensor fusion code.

The objective of the TESSA program is to enhance the F-15E's capability to locate, identify, and destroy stationary and moving threat TMD assets. To accomplish this, the radar improvements include enhanced SAR resolution (4' x 6' maps), detecting probable ground targets, and overlaying the most likely targets on the SAR map display. A new Fused Feature Automatic Target Recognition (FFATR) processor will be installed on the aircraft to host the sensor fusion software. A new interface will be provided between the radar and the FFATR in order for the radar to cue the FLIR. In addition, the code in the F-15E Central Computer will be modified to support the additional TESSA functionality.

The TESSA radar processing detects an area of interest (i.e., an object that is likely to be man-made) in a radar map. In addition, the radar estimates the size, shape, and orientation of the object. The area of interest is sent to the fusion engine to cue the FLIR. This description suggests several points where checkers might be used. For example, checking the radar data used to cue the FLIR ensures that the radar is improving, rather than degrading the FLIR performance.

Three to six checkers are being added to the automatic target cueing (ATC) mode of the APG-70 radar software. Although the radar ATC algorithms were previously validated in a laboratory demonstration using radar maps and FLIR images recorded during flight test, the implementation of those algorithms in the APG-70 operational software is all new code. This experiment is currently ongoing and results will be published in the SCEISS interim report for 1999.

*Continued on page 22*

## Self-Checking Systems

*Continued from page 21*

### Summary and Conclusions

Under the SCEISS program, we are making progress towards demonstrating the advantages of self-checking systems and transitioning self-checking system technology from theory to practice. We took advantage of a production radar tape upgrade that was starting software testing and proceeding into flight test during the first year of the SCEISS program. We implemented a checker for deep system performance to detect suspicious behavior that is often overlooked during traditional testing, but which indicates that significant performance degradation could occur after the system is deployed. As a result, suspicious behavior was observed and reported back to the regular program development team for further analysis and correction of the problem. From this experiment, we collected evidence that effective checkers can be implemented at a relatively small additional cost with minor runtime overhead to the software being checked.

We also identified candidate checkers for the TESSA IV technology demonstration program that have the potential for reducing risk as the program proceeds into laboratory and flight test in 1999. We are in the process of implementing those checkers and exercising them in the integration laboratory and in flight test. The metrics collected during the TESSA IV demonstration will provide further evidence of the cost effectiveness of using self-checking techniques to improve the quality and reliability of embedded information systems.

In out years of the SCEISS program, we are planning further demonstration programs using checker technology. We believe that checkers will prove especially valuable in space systems where reliability is especially important.

### Author Contact Information

**Tod J. Reinhart**
AFRL/IFTA
Embedded Information Systems
2241 Avionics Circle, Suite 32
WPAFB OH 45433-7334
Tod.Reinhart@sensors.wpafb.af.mil
(937) 255-6548 ext. 3582
DSN 785-6548 x3582
Fax: (937) 656-4277
DSN-Fax 656-4277

**D. Joel Mellema, and**
**Carolyn Boettcher**
Raytheon Systems Company
www.raytheon.com/rsc/

### References

[1] Butler, R., Finelli, G., "The Infeasibility of Quantifying the Reliability of Life-Critical Real-Time Software", IEEE Transactions on Software Engineering, Vol. 19, No. 1, January, 1993.

[2] Blum, M., Kannan, S. "Designing Programs that Check their Work", Proceedings of the ACM 21st Annual Symposium on the Theory of Computing, May, 1989.

[3] Wasserman, H., Blum, M., "Software Reliability via Run-Time Result-Checking", Journal of the ACM, Vol. 44, No. 6, November, 1997.

[4] Shreve, D., Mellema, D. J., Boettcher, C., "Real-time Checkers: Built-in Test for Mission Critical Software", IEEE/AIAA Digital Avionics Systems Conference Proceedings, Vol. I, October 26-30, 1997.

[5] Boettcher, C., Mellema, D. J., "Program Checkers: Practical Applications to Real-Time Software", Test Facility Working Group Conference, 1995.

[6] Blum, M., Wasserman, H., "Reflections on the Pentium Division Bug", IEEE Transactions on Computers, Vol. 45, No. 4, April, 1996.

[7] Reinhart, T., Boettcher, C., Wasserman, H., "An Automated Testing Methodology Based on Self-Checking Software", NAECON Proceedings, IEEE, July, 1998.

[8] Self-Checking Embedded Information System Software Interim Report, December 1997 - November 1998, Raytheon Systems Company, Los Angeles, CA.

## DACS Technical Area Task (TAT), “Science and Technology Corporate Information Management Support,” Wins the Coveted Hammer Award

### What is Hammer?

The Hammer Award is presented to teams of federal employees who have made significant contributions in support of reinventing government principles. The award is the Vice President’s answer to yesterday’s government and its $400 hammer. Fittingly, the award consists of a $6.00 hammer, a ribbon, and a note from Vice President Gore, all in an aluminum frame. More than 1,000 Hammer Awards have been presented to teams comprised of federal employees, state and local employees, and citizens who are working to build a better government.

### The Need … Producing the Annual DoD In-House RDT&E Activities Report

Preparation and publication of the annual DoD In-House Research, Development, Test and Evaluation (RDT&E) Activities Report, was identified as an effort requiring process improvement, by the Executive Working Group (EWG) established by the Director of Defense Research and Engineering (DDR&E), to steer the Science and Technology (S&T) Business Process Reengineering (BPR) program. The report has been produced in official form for the DDR&E since 1966, and is the DDR&E’s central source of information on laboratory status.

In the past, the document was generated in hard copy form, using traditional data collection methods, taking up to two years to complete the publication cycle from initiation of the data call to printing and distribution of hard copies. This process severely degraded the value of the report, primarily due to the perishable nature of the information it contained.

To guide the process improvement initiative, the EWG further established an In-House Functional Improvement Team (IH FIT), consisting of members from each of the Services, the Deputy Director of Defense Research and Engineering for Lab Management and Technology Transition (DTSE&E LM&TT), Director, Test, Systems Engineering and Evaluation (DTSE&E), the Uniformed Services University of Health Sciences (USUHS), and the S&T BPR office. **The team’s charter was to reduce the time and cost of publishing the report by reinventing the current process and exploiting the latest technologies.**

The Web based process developed by the DACS dramatically streamlined the report publication cycle:

- The time to publication was reduced by over 75%;
- The cost of producing the report was lowered by over 50%; and
- Work hours spent by those producing the report were reduced by almost 60%.

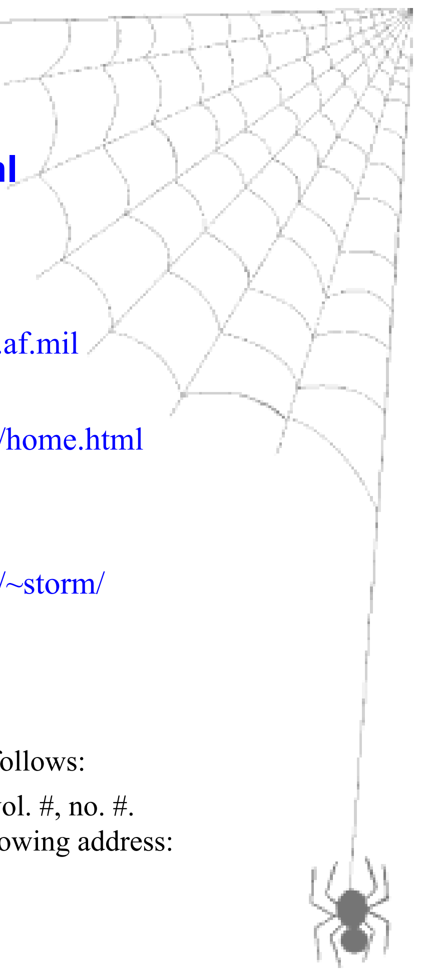
The DACS is a DoD Information Analysis Center (IAC) sponsored by DTIC that focuses on software technology. The DACS serves as a central source for current and readily usable information concerning software technology and development, and provides technical assistance and solutions to user-specified problems through TATs.

**Mr. Dan Snell**
DACS Deputy Director
P.O. Box 1400
Rome, NY 13442–1400
(800)214-7921
Fax (315) 334-4964
dan.snell@ssc.de.ittind.com

**A special thanks to Morton Hirschberg, newly elected Editorial Board Chairman**

## Software Tech News on the World Wide Web

**This newsletter, including referenced full-length articles, is available on the web at:**
**www.dacs.dtic.mil/awareness/newsletters/listing.shtml**



## Other Software Testing On-line Resources

DoD DACS Software Testing Topic Area - www.dacs.dtic.mil

AFOTEC Air Force Operational Test and Evaluation Center - www.afotec.af.mil

Director, Test, Systems Engineering & Evaluation - www.acq.osd.mil/te/

IEEE Test Technology, Technical Council - www.computer.org/tab/tttc/tac/home.html

Software Research Institute- www.soft.com

Software Testing Institute - www.ondaweb.com/sti/

STORM:  A nexus of Software Testing Online Resources - www.mtsu.edu/~storm/

STSC Software Testing Page - www.stsc.hill.af.mil/swtesting/index.asp

## Article Reproduction

**DoD Data & Analysis Center for Software**
**P.O. Box 1400**
**Rome, NY 13442-1400**

First-Class Mail
U.S. Postage
**P A I D**
Colo. Spgs., CO
**Permit No. 745**

*Return Service Requested*

# DoD DACS Products & Services Order Form

| | | |
|---|---|---|
| Name: | Position/Title: | |
| Organization: | | Acronym: |
| Address: | | |
| City: | State: | Zip Code: |
| Country: | E-mail: | |
| Telephone: | Fax: | |

| *Product Description* | *Format* | *Quantity* | *Price* | *Total* |
|---|---|---|---|---|
| **The DACS Information Package** | *Note: All Disks are available in PC or Mac | | | |
| ❑ Including: 2 recent Software Tech News newsletters, and several DACS Products & Services Brochures | Documents | | FREE | FREE |
| **Empirical Data** | | | | |
| ❑ Architecture Research Facility (ARF) Error Dataset | Disk | | $ 50 | |
| ❑ NASA / Software Engineering Laboratory (SEL) Dataset | CD-ROM | | $ 50 | |
| ❑ NASA / AMES Error/Fault Dataset | Disk | | $ 50 | |
| ❑ Software Reliability Dataset | Disk | | $ 50 | |
| ❑ DACS Productivity Dataset | Disk | | $ 50 | |
| **Technical Reports** | | | | |
| ❑ A Business Case for Software Process Improvement | Document | FREE with Spreadsheet → | $ 25 | |
| ❑ ROI from Software Process Improvement Spreadsheet | Disk | | $ 40 | |
| ❑ A History of Software Measurement at Rome Laboratory | Document | | $ 25 | |
| ❑ An Analysis of Two Formal Methods: VDM and Z | Document | | $ 25 | |
| ❑ An Overview of Object-Oriented Design | Document | | $ 25 | |
| ❑ Artificial Neural Networks Technology | Document | | $ 25 | |
| ❑ A Review of Formal Methods | Document | | $ 25 | |
| ❑ A Review of Non-Ada to Ada Conversion | Document | | $ 25 | |
| NEW! ❑ Using Defect Tracking & Analysis to Improve Software Qual | Document | | $ 50 | |
| ❑ Software Design Methods | Document | | $ 25 | |
| ❑ Distributable Database Technology | Document | | $ 25 | |
| ❑ Electronic Publishing on the World Wide Web: An Engineering Approach | Document | SALE Item! → | **$ 5** | |
| NEW! ❑ Object Oriented Database Management Systems (Revisited) | Document | | $ 50 | |
| ❑ Software Analysis and Testing Technologies | Document | | $ 25 | |
| ❑ Software Design Methods | Document | | $ 25 | |
| ❑ Software Prototyping and Requirements Engineering | Document | | $ 25 | |
| ❑ Software Interoperability | Document | | $ 25 | |
| ❑ Software Reusability | Document | | $ 25 | |
| NEW! ❑ Understanding & Improving Technology Transfer in Soft Eng | Document | | $ 50 | |
| **Bibliographic Products** | | | | |
| ❑ Rome Laboratory Research in Software Measurement | Document | | $ 25 | |
| ❑ DACS Custom Bibliographic Search | Disk | | $ 40 | |
| ❑ DACS Software Engineering Bibliographic Database (SEBD) | CD-ROM | | $ 50 | |

**Method of Payment:**

❑ Check ❑ Mastercard ❑ Visa

*Number of Items Ordered* ☐ *Total Cost* ☐

Credit Card # ______________________ Expiration Date ______________________

Name on Credit Card ______________________ Signature ______________________

Mail this form or: Phone: (315) 334-4905, Fax: (315) 334-4964
E-mail: **cust-liasn@dacs.dtic.mil**

**This form is also on-line at: www.dacs.dtic.mil/forms/orderform.shtml**

---fold here---

---fold here---

Fix
postage
here

DoD Data & Analysis Center for Software
Attn: DACS Customer Liaison
PO. Box 1400
Rome, NY 13442-1400